The Simple Economics of Open Source[*]

Josh Lerner[**] and Jean Tirole[***]

February 25, 2000

There has been a recent surge of interest in open source software development, which involves developers at many different locations and organizations sharing code to develop and refine programs. To an economist, the behavior of individual programmers and commercial companies engaged in open source projects is initially startling. This paper makes a preliminary exploration of the economics of open source software. We highlight the extent to which labor economics, especially the literature on "career concerns," can explain many of these projects' features. Aspects of the future of open source development process, however, remain somewhat difficult to predict with "off-the-shelf" economic models.

[*] The assistance of the Harvard Business School's California Research Center, and Chris Darwall in particular, was instrumental in the development of the case studies and is gratefully acknowledged. We also thank a number of practitioners—especially Eric Allman, Mike Balma, Brian Behlendorf, Keith Bostic, Tim O'Reilly, and Ben Passarelli—for their willingness to generously spend time discussing the open source movement. Jacques Crémer, Bernard Salanié, and Rob Merges provided helpful comments. Harvard Business School's Division of Research provided financial support. The Institut D'Economie Industrielle receives research grants from a number of corporate sponsors, including Microsoft Corporation. All opinions and errors, however, remain our own.

[**] Harvard Business School and NBER.

[***] IDEI and GREMAQ (CNRS UMR 5604), Toulouse, CERAS (CNRS URA 2036), and MIT.

1. Introduction

Over the past two years, there has been a surge of interest in open source software development. Interest in this process, which involves software developers at many different locations and organizations sharing code to develop and refine software programs, has been spurred by three factors:

- *The rapid diffusion of open source software*. A number of open source products, such as the Apache web server, dominate their product categories. In the personal computer operating system market, International Data Corporation estimates that the open source program Linux has between seven to twenty-one million users worldwide, with a 200% annual growth rate. Many observers believe it represents a leading potential challenger to Microsoft Windows in this important market segment.
- *The significant capital investments in open source projects*. Over the past two years, numerous major corporations, including Hewlett Packard, IBM, and Sun, have launched projects to develop and use open source software. Meanwhile, a number of companies specializing in commercializing Linux, such as Red Hat and VA Linux, have completed initial public offerings, and other open source companies such as Cobalt Networks, Collab.Net, Scriptics, and Sendmail have received venture capital financing.
- *The new organization structure*. The collaborative nature of open source software development has been hailed in the business and technical press as an important organizational innovation.

Yet to an economist, the behavior of individual programmers and commercial companies engaged in open source processes is startling. Consider these quotations by two leaders of the open source community:

> The idea that the proprietary software social system—the system that says you are not allowed to share or change software—is unsocial, that it is unethical, that it is simply wrong may come as a surprise to some people. But what else can we say about a system based on dividing the public and keeping users helpless? [Stallman, 1999]

> The "utility function" Linux hackers is maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. [Parenthetical comment deleted] Voluntary cultures that work this way are actually not uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom explicitly recognizes "egoboo" (the enhancement of one's reputation among other fans) [Raymond, 1999b].

It is not initially clear how these claims relate to the traditional view of the innovative process in the economics literature. Why should thousands of top-notch programmers contribute freely to the provision of a public good? Any explanation based on altruism[1] only goes so far. While users in less developed countries undoubtedly benefit from access to free software, many beneficiaries are well-to-do individuals or Fortune 500 companies. Furthermore, altruism has not played a major role in other industries, so it would have to be explained why individuals in the software industry are more altruistic than others.

This paper seeks to address this puzzle, by making a preliminary exploration of the economics of open source software. Reflecting the early stage of the field's development, we do not seek to develop new theoretical frameworks or to statistically analyze large samples. Rather, we focus on three "mini-cases" of particular projects:

[1] The media like to portray the open source community as wanting to help mankind, as it makes a good story. Many open source advocates put limited emphasis on this explanation.

Apache, Perl, and Sendmail.[2] We seek to draw some initial conclusions about the key economic patterns that underlie the open source development of software. We find that much can be explained by reference to economic frameworks. We highlight the extent to which frameworks of labor economics, and in particular the literature on "career concerns," can explain many of the features of open source projects.

At the same time, we acknowledge that aspects of the future of open source development process remain somewhat difficult to predict with "off-the-shelf" economic models. In the final section of this paper, we highlight a number of puzzles that the movement poses. It is our hope that this section will have itself an "open source" nature: that it will stimulate research by other economic researchers as well.

Finally, it is important to acknowledge the relationship with the earlier literature on technological innovation. The open source development process is somewhat reminiscent of the type of "user-driven innovation" seen in many other industries. Among other examples, Rosenberg's [1976] studies of the machine tool industry and von Hippel's [1988] of scientific instruments have highlighted the role that sophisticated users can play in accelerating technological progress. In many instances, solutions developed by particular users for individual problems have become more general solutions for wide classes of users. But as we shall argue below, certain aspects of the open source process—especially the extent to which contributors' work is recognized and rewarded—are quite distinct from earlier settings.

2. The Nature of Open Source Software

[2] These are summarized in Darwall and Lerner [2000].

While media attention to the phenomenon of open source software has been recent, the basic behaviors are much older in their origins. There has long been a tradition of sharing and cooperation in software development. But in recent years, both the scale and formalization of the activity have expanded dramatically with the widespread diffusion of the Internet.[3] In the discussion below, we will highlight three distinct eras of cooperative software development.

2.1 The first era: early 1960s to the early 1980s.

Many of the key aspects of the computer operating systems and the Internet were developed in academic settings such as Berkeley and MIT during the 1960s and 1970s, as well as in central corporate research facilities where researchers had a great deal of autonomy (such as Bell Labs and Xerox's Palo Alto Research Center). In these years, the sharing by programmers in different organizations of basic operating code of computer programs—the source code—was commonplace.[4]

Many of the cooperative development efforts in the 1970s focused on the development of an operating system that could run on multiple computer platforms. The most successful examples, the Unix operating system and the C language used for developing Unix applications, were originally developed at AT&T's Bell Laboratories. The software was then installed across institutions, being transferred freely or for a nominal charge. Many of the sites where the software was installed made further innovations, which were

[3] This history is of necessity highly abbreviated. For more detailed treatments, see Browne [1999], DiBona, Ockman, and Stone [1999], Gomulkiewicz [1999], Levy [1984], and Raymond [1999a].

[4] Programmers write source code using languages such as Basic, C, and Java. By way of contrast, most commercial software vendors only provide users with object, or binary, code. This is the sequence of 0s and 1s that directly communicates with the computer, but which is difficult for programmers to interpret or modify. When the source code is made available to other firms by commercial developers, it is typically licensed under very restrictive conditions.

in turn shared with others. The process of sharing code was greatly accelerated with the diffusion of Usenet, a computer network begun in 1979 to link together the Unix programming community. As the number of sites grew rapidly (e.g., from 3 in 1979 to 400 in 1982), the ability of programmers in university and corporate settings to rapidly share technologies was considerably enhanced.

These cooperative software development projects were undertaken on a highly informal basis. Typically no effort to delineate property rights or to restrict reuse of the software were made. This informality proved to be problematic in the early 1980s, when AT&T began enforcing its (purported) intellectual property rights related to Unix.

2.2 The second era: early 1980s to the early 1990s.

In response to these threats of litigation, the first efforts to formalize the ground rules behind the cooperative software development process emerged. This ushered in the second era of cooperative software development. The critical institution during this period was the Free Software Foundation, begun by Richard Stallman of the MIT Artificial Intelligence Laboratory in 1983. The foundation sought to develop and disseminate a wide variety of software without cost.

One important innovation introduced by the Free Software Foundation was a formal licensing procedure that aimed to preclude the commercialization of cooperatively developed software. In exchange for being able to use and modify the GNU software (as it was known), users had to agree to make the source code freely available (or at a nominal cost). As part of the General Public License (GPL, also known as "copylefting"), the user had to also agree not to impose licensing restrictions on others. Furthermore, all enhancements to the code—and even code that intermingled the

cooperatively developed software with that developed separately—had to be licensed on the same terms. It is these contractual terms that distinguish open source software from shareware (where the binary files but not the underlying source code are made freely available, possibly for a trial period only) and public-domain software (where no restrictions are placed on subsequent users of the source code).[5]

This project, as well as contemporaneous efforts, also developed a number of important organizational features. In particular, these projects employed a model where contributions from many developers were accepted (and frequently publicly disseminated or posted). The right to modify the official version of the program, however, was confined to a smaller subset of individuals closely involved with the project, or in some cases, an individual leader. In some cases, the project's founder (or his designated successor) served as the leader; in others, leadership rotated between various key contributors.

2.3 The third era: early 1990s to today.

The widespread diffusion of Internet access in the early 1990s led to a dramatic acceleration of open source activity. The volume of contributions and diversity of contributors expanded sharply, and numerous new open source projects emerged, most notably Linux (a variant of the UNIX operating system developed by Linus Torvalds in 1991). As discussed in detail below, interactions between commercial companies and the open source community also became commonplace in the 1990s.

[5] It should be noted, however, that some projects, such as the Berkeley Software Distribution (BSD) effort, did take alternative approaches during the 1980s. The BSD license is much less constraining than the GPL: anyone can modify the program and redistribute it for a fee without making the source code freely available. In this way, it was a continuation of the university-based tradition of the 1960s and 1970s.

Another innovation during this period was the proliferation of alternative approaches to licensing cooperatively developed software. During the 1980s, the GPL was the dominant licensing arrangement for cooperatively developed software. This changed considerably during the 1990s. In particular, Debian, an organization set up to disseminate Linux, developed the "Debian Social Contract" in 1995. This agreement allowed licensees greater flexibility in using the program, including the right to bundle the cooperatively developed software with proprietary code. This more flexible licensing arrangement was adopted in early 1997 by a number of individuals involved in cooperative software development, and was subsequently known as the "Open Source Definition." As the authors explained:

> **License Must Not Contaminate Other Software.** The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software. Rationale: Distributors of open-source software have the right to make their own choices about their own software [Open Source Initiative, 1999].

Unlike the General Public License, this new license is not "viral": it does not "infect" all code that was bundled with the software with the requirement that it be covered under the license agreement as well.

The past few years have seen unprecedented growth of open source software. At the same time, the movement has faced a number of challenges. We will highlight two of these here: the "forking" of projects (the development of competing variations) and the development of products for high-end users.

One issue that has emerged in a number of open source projects is the potential for programs splintering into various variants. In some cases, passionate disputes over product design have led to the splintering of open source projects into different variants.

Examples of such splintering are the Berkeley Unix program and Sendmail during the late 1980s.

Another challenge has been the apparently lesser emphasis on documentation and support, user interfaces,[6] and backward compatibility found in at least some open source projects. The relative technological features of software developed in open source and traditional environments are a matter of passionate discussion. Some members of the community believe that this production method dominates traditional software development in all respects. But many open source advocates argue that open source software tends to be geared to the more sophisticated users.[7] This point is made colorfully by one open source developer:

> [I]n every release cycle Microsoft always listens to its *most ignorant customers*. This is the key to dumbing down each release cycle of software for further assaulting the non-personal computing population. Linux and OS/2 developers, on the other hand, tend to listen to their *smartest* customers… The good that Microsoft does in bringing computers to non-users is outdone by the curse that they bring on experienced users [Nadeau, 1999].

Certainly, the greatest diffusion of open source projects appears to be in settings where the end users are sophisticated, such as the Apache server installed by systems administrators. In these cases, users are apparently more willing to tolerate the lack of detailed documentation or easy-to-understand user interfaces in exchange for the cost savings and the possibility of modifying the source code themselves. In several projects, such as Sendmail, project administrators chose to abandon backward compatibility in the

[6] Two main open source projects (GNOME and KDE) are meant to remedy Linux's handicap on the desktop (mouse and windows interfaces).

[7] For example, Torvalds [1999] argues that the Linux model works best with developer-type software. Ghosh [1999] views the open source process as a large repeated game process of give-and-take among developer-users (the "cooking pot" model).

interests of preserving program simplicity. One of the rationales for this decision was that administrators using the Sendmail system were responsive to announcements that these changes would be taking place, and rapidly upgraded their systems. In a number of commercial software projects, it has been noted, these types of rapid responses are not as common. Once again, this reflects the greater sophistication and awareness of the users of open source software.

The debate about the ability of open source software to accommodate high-end users' needs has direct implications for the choice of license. The recent popularity of more liberal licenses such as the Debian Social Contract and the Open Source Definition and the concomitant decline of the GNU license are related to the rise in the "pragmatists'" influence. These individuals believe that allowing proprietary code and for-profit activities in segments that would otherwise be poorly served by the open-source community will provide the movement with its best chance for success.

3. The Origins of the Three Programs

Each of the three case studies was developed through the review of printed materials and interviews (as well as those posted on various web sites) and face-to-face meetings with one or more key participants in the development effort. In addition, we held a number of conversations with knowledgeable observers of the open source movement. In Sections 4 and 5, we will frequently draw on examples from the three cases. Nonetheless, we felt it would be helpful to first provide a brief overview of the three development projects.

3.1 Apache.

The development of Apache began in 1994. Brian Behlendorf, then 21, had the responsibility for operating one of the first commercial Internet servers in the country, that powering *Wired* magazine's HotWired web site. This server, like most others in the country, was at the time running the Unix-based software written at the National Center for Supercomputer Applications (NCSA) at the University of Illinois. (The only competitive product at the time was the server developed at the joint European particle physics research facility CERN.) The NCSA had distributed its source code freely and had a development group actively involved in refining the code in consultation with the pioneering users. As Behlendorf and other users wrote emendations, or "patches," for the NCSA server, they would post them as well to mailing lists of individuals interested in Internet technology.

Behlendorf and a number of other users, however, encountered increasing frustrations in getting the NCSA staff to respond to their suggestions. (During this time, a number of the NCSA staff had departed to begin Netscape, and the University was in the process of negotiating a series of licenses of its software with commercial companies.) As a result, he and six other pioneering developers decided to establish a mailing list to collect and integrate the patches to the NCSA server software. They agreed that the process would be a collegial one. While a large number of individuals would be able to suggest changes, only a smaller set would be able to actually make changes to the physical code. In August 1995, the group released Apache 0.8, which represented a substantial departure from earlier approaches. A particular area of revision was the Application Program Interface (API), which allowed the development of Apache features to be very

"modular." This step enabled programmers to make contributions to particular areas without affecting other aspects of the programs.

The diffusion of Apache has been quite dramatic. Periodic surveys by Netcraft show that the share of publicly observable Web servers (*i.e.*, those not behind firewalls) running Apache rose from 31% in April 1996 to 44% in June 1997 to 55% in September 1999.[8] In 1999, the Apache Software Foundation was established to oversee the development and diffusion of the program. The current status of Apache, as well as the other two open source projects that we focused on, is summarized in **Table 1**.

3.2 Perl.

Perl, or the Practical Extraction and Reporting Language, was created by Larry Wall in 1987. Wall, a programmer with Burroughs (a computer mainframe manufacturer now part of Unisys) had already written a number of widely adopted software programs. These included a program for reading postings on on-line newsgroups and a program that enabled users to readily update old source code with new patches.

The specific genesis of Perl was the large number of repetitive system administration tasks that Wall was asked to undertake while at Burroughs. In particular, Wall was required to synchronize and generate reports on two Unix-based computers as part of a project that Burroughs was undertaking for the U.S. National Security Agency. He realized that there was a need for a program language that was somewhere between the Unix shell language and the C language (suitable for developing complex programming applications). The Perl language sought to enable programmers to rapidly undertake a

[8] A complication is introduced by the fact that firewall-protected servers may be quite different in nature. For instance, a survey of both protected and unprotected servers in the summer of 1996 by Zoma Research concluded that open source server programs, including Apache, accounted for only 7% of all installations, far less than the contemporaneous Netcraft estimate.

wide variety of system administration tasks. The program was first introduced in 1987 via the Internet. It has become widely accepted as a language for developing scripts for Apache web servers, and is incorporated in a number of other programs.

Perl is administered on a rotating basis: the ten to twenty programmers (the number fluctuates over time) who have been most actively involved in the program take turns managing different aspects of the project. Wall himself has joined the staff of O'Reilly & Associates, a publisher specializing in manuals documenting open source programs. While he is no longer actively contributing to the programming, he remains active in managing the project.

Two efforts to establish a Perl-related foundation have foundered. The Perl Institute had been intended to ensure that less glamorous tasks, such as documentation, were undertaken, in order to enhance the long-run growth of Perl. The failure of these efforts may have reflected more about the specifics of the individual personalities involved than the prospects of the program itself.

3.3 *Sendmail.*

Sendmail was originally developed in the late 1970s by Eric Allman, a graduate student in computer science at the University of California at Berkeley. As part of his responsibilities, Allman worked on a variety of software development and system administration tasks at Berkeley.

One of the major challenges that Allman faced was the incompatibility of the two major computer networks on campus. The approximately one dozen Unix-based computers had been originally connected through "BerkNet," a locally developed program that provided continuous interconnection. These computers, in turn connected

to those on other campuses through telephone lines, using the UUCP protocol (Unix-to-Unix Copy Protocol). Finally, the Arpanet, the direct predecessor to the Internet, was introduced on the Berkeley campus around this time. Each of the networks used a different communications protocol: for instance, each person had multiple e-mail addresses, depending on the network from which the message was sent. To cope with this problem, Allman developed in 1979 a program called "Delivermail," which provided a way to greatly simplify the addressing problem. In an emendated form that allowed it to address a large number of domains, it was released two years later as "Sendmail."

Sendmail was soon adopted as the standard method of routing e-mail on the Arpanet. As the network grew, however, its limitations became increasingly apparent. A variety of enhanced versions of Sendmail were released in the 1980s and early 1990s which were incompatible with each other—in the argot of the open source community, the development of the program "forked." In 1993, Allman, who had returned to working at Berkeley after being employed at a number of software firms, undertook a wholesale rewrite of Sendmail. The development was sufficiently successful that the incompatible versions were largely abandoned in favor of the new version. By 1998, it was estimated that 80% of all e-mail traffic was sent by Sendmail.

In 1997, Allman established Sendmail, Inc. The company, which has been financed by a leading venture capital group Benchmark Capital, is seeking to sell Sendmail-related software enhancements (such as more user-friendly interfaces) and services. At the same time, the company seeks to encourage the continuing development of the software on an open source basis. For instance, Sendmail, Inc. employs two engineers who work almost

full time on contributions to the open source program, which is run by the non-profit Sendmail Consortium.

4. What Does Economic Theory Tell Us about Open Source?

4.1 What motivates programmers? Theory.

A programmer participates in a project, whether commercial or open source, only if she derives a net benefit (broadly defined) from engaging in the activity. The net benefit is equal to the *immediate* payoff (current benefit minus current cost) plus the *delayed* payoff.

A programmer working on a software development project incurs a variety of immediate benefits and costs. First, the programmer receives monetary compensation if she is working for a commercial firm. Second, the programmer may be fixing a bug or customizing a program for her own benefit (as well as, in the case of an open source process, for the benefit of others.) Third, the programmer incurs an opportunity cost of her time. While she is working on this project, she is unable to engage in another programming activity. The actual cost of this time depends on how enjoyable the work is.

The delayed reward covers two distinct, although hard-to-distinguish, incentives. The *career concern incentive* refers to future job offers, shares in commercial open source-based companies,[9] or future access to the venture capital market. The *ego gratification incentive* stems from a desire for peer recognition. Probably most programmers respond

[9] Linus Torvalds and others have been awarded shares in Linux-based companies that went public. Most certainly, these rewards were unexpected and did not affect the motivation of open source programmers. If this practice becomes "institutionalized," such rewards will in the future be expected and therefore impact the motivation of open source leaders.

to both incentives. There are some differences between the two. The programmer mainly preoccupied by peer recognition may shun future monetary rewards, and may also want to signal her talent to a slightly different audience than those motivated by career concerns. From an economic perspective, however, the incentives are similar in most respects. We will group the career concern incentive and the ego gratification incentive under a single heading: the *signaling incentive*.

Economic theory [e.g., Holmström, 1999] suggests that this signaling incentive is stronger,

a) the more visible the performance to the relevant audience (peers, labor market, venture capital community),

b) the higher the impact of effort on performance, and

c) the more informative the performance about talent.

The first condition gives rise to what economists call "strategic complementarities." To have an "audience," programmers will want to work on software projects that will attract a large number of other programmers. This suggests the possibility of multiple equilibria. The same project may attract few programmers because programmers expect that other programmers will not be interested; or it may flourish as programmers (rationally) have faith in the project.

The same point applies to forking in a given open source project. Open source processes are in this respect quite similar to academic research. The latter is well known to exhibit fads. Fields are completely neglected for years, while others with apparently no superior intrinsic interest attract large numbers of researchers. Fads in academia are frowned upon for their inefficient impact on the allocation of research. It should not be

ignored, however, that fads also have benefits. A fad can create a strong signaling incentive: researchers working in a popular area may be highly motivated to produce a high-quality work, since they can be confident that a large audience will examine their work.

4.2 *Comparison between open source and closed source programming incentives.*

To compare programmers' incentives in the open source and proprietary settings, we need to examine how the fundamental features of the two environments shape the incentives just reviewed. We will first consider the relative short-term rewards, and then turn to the deferred compensation.

Commercial projects have an edge on the current-compensation dimension because the proprietary nature of the code generates income. This makes it privately worthwhile for private companies to offer salaries.[10] This contention is the old argument in economics that the prospect of profit encourages investment, which is used, for instance, to justify the awarding of patents to encourage invention.

By way of contrast, an open source project may well lower the cost for the programmer, for two reasons:

i) *"Alumni effect":* Because the code is freely available to all, it can be used in schools and universities for learning purposes; so it is already familiar to programmers. This reduces their cost of programming for UNIX, for example.[11]

[10] To be certain, commercial firms (e.g., Netscape, Sun, O'Reilly, Transmeta) supporting open source projects are also able to compensate programmers, because they indirectly benefit financially from these projects. Similarly, the government and not-for-profit corporations have done some subsidizing of open source projects. Still, there should be an edge for commercial companies.

[11] WhiIe we are here interested in private incentives to participate, note that this complementarity between apprenticeship and projects is socially beneficial. The social benefits might not increase linearly with open source market share, however, since the competing open source projects may end up competing for attention in the same common pool of students.

ii) *Customization and bug-fixing benefits:* The cost of contributing to an open source project is lower if the activity brings about a private benefit (bug fixing, customization) for the programmer and her firm. Note again that this factor of cost reduction is directly linked to the openness of the source code.

Let us now turn to the delayed reward (signaling incentive) component. In this respect too, the open source process has some benefits over the closed source approach. As we noted, signaling incentives are stronger, the more visible the performance and the more attributable the performance to a given individual. Signaling incentives therefore may be stronger in the open source mode for three reasons:

i) *Better performance measurement*: Outsiders can only observe inexactly the functionality and/or quality of individual elements of a typical commercially developed program, as they are unable to observe the proprietary source code. By way of contrast, in an open source project, the outsiders are able to see not only what the contribution of each individual was and whether that component "worked," but also whether the task was hard, if the problem was addressed in a clever way, whether the code can be useful for other programming tasks in the future, and so forth.

ii) *Full initiative:* The open source programmer is her own boss and takes full responsibility for the success of a subproject. In a hierarchical commercial firm, however, the programmer's performance depends on her supervisor's interference, advice, *etc*. Economic theory would predict that the programmer's performance is more precisely measured in the former case.

iii) *Greater fluidity:* It may be argued that the labor market is more fluid in an open source environment. Programmers are likely to have less idiosyncratic, or firm-specific, human capital that limits shifting one's efforts to a new program or work environment. (Since many elements of the source code are shared across open source projects, more of the knowledge they have accumulated can be transferred to the new environment).

These theoretical arguments also provide insights as to *who* is more likely to contribute and *what tasks* are best suited to open source projects. Sophisticated users derive direct benefits when they customize or fix a bug in open source software.[12] A second category of potential contributors consists of individuals with strong signaling incentives; these may use open source software as a port of entry. For instance, open source processes may give a talented system administrator at a small academic institution (who is also a user!) a unique opportunity to signal her talent to peers, prospective employers, and the venture capital community.[13]

[12] A standard argument in favor of open source processes is their massive parallel debugging. Typically, commercial software firms can only ask users to point at problems: beta testers do not fix the bugs, they just report them. It is also interesting to note that many commercial companies do not discourage their employees from working on open source projects. In many cases where companies encourage such involvement, programmers use open source tools to fix problems. Johnson [1999] builds a model of open source production by a community of user-developers. There is one software program or module to be developed, which is a public good for the potential developers. Each of the potential developers has a private cost of working on the project and a private value of using it; both of which are private information. Johnson shows that the probability that the innovation is made need not increase with the number of developers, as free-riding is stronger when the number of potential developers increases.

[13] An argument often heard in the open source community is that people participate in open source projects because programming is fun and because they want to be "part of a team." While this argument may contain a grain of truth, it is puzzling as it stands; for, it is not clear why programmers who are part of a commercial team could not enjoy the same intellectual challenges and the same team interaction as those engaged in open source development. The argument may reflect the ability of programmers to use participation in open source projects to overcome labor market rigidities that make signaling in other ways problematic.

As to the tasks that may appeal to the open source community, one would expect that tasks such as those related to the operating systems and programming languages, whose natural audience is the community of programmers, would give rise to strong signaling incentives. By way of contrast, tasks aiming at helping the much-less-sophisticated end user—e.g., documentation, design of easy-to-use interfaces, technical support, and insuring backward compatibility—usually provide lower signaling incentives.[14]

4.3 *Evidence on individual incentives*.

A considerable amount of evidence is consistent with an economic perspective.

First, user benefits are key to a number of open source projects. One of the origins of the free software movement was Stallman's inability to improve a printer program because Xerox refused to release the source code. In each of the three scenarios described in section 3, the project founders were motivated by information technology problems that they had encountered in their day-to-day work. For instance, in the case of Apache, the initial set of contributors was almost entirely system administrators who were struggling with the same types of problems as Behlendorf. In each case, the initial release was "runnable and testable": it provided a potential, even if imperfect, solution to a problem that was vexing considerable numbers of data processing professionals.

Second, it is clear that giving credit to authors is essential in the open source movement. This principle is included as part of the nine key requirements in the "Open Source Definition" [Open Source Initiative, 1999]. This point is also emphasized by

[14] Valloppillil [1998] further argues that reaching commercial grade quality often involves unglamorous work on power management, management infrastructure, wizards, *etc*., that makes it unlikely to attract open source developers.

Raymond [1999b], who points out "surreptitiously filing someone's name off a project is, in cultural context, one of the ultimate crimes."

Finally, the reputational benefits that accrue from successful contributions to open source projects appear to have real effects on the developers. This is acknowledged within the open source community itself. For instance, Raymond [1999b] notes three primary benefits that accrue to successful contributors of open source projects "good reputation among one's peers, attention and cooperation from others, … [and] higher status [in the] … exchange economy." Thus, while some of benefits conferred from participation in open source projects may be less concrete in nature, there also appear be quite tangible—if delayed—rewards.

The Apache project provides a good illustration of these observations. The project makes a point of recognizing all contributors on its web site, even those who simply identify a problem without proposing a solution. Similarly, the organization highlights its most committed contributors, who have the ultimate control over the project's evolution. Moreover, it appears that many of the skilled Apache programmers have benefited materially from their association with the organization. Numerous contributors have been hired into Apache development groups within companies such as IBM, become involved in process-oriented companies such as Collab.Net which seek to make open source projects more feasible (see below), or else moved into other Internet tools companies in ways that were facilitated by their expertise and relationships built up during the open source movement. Meanwhile, many of the new contributors are already employed by corporations, and working on Apache development as part of their regular assignments.

There is also substantial evidence that open source work may be a good stepping stone for securing access to venture capital. For example, the founders of Sun, Netscape, and Red Hat had signaled their talent in the open source world. In **Table 2**, we summarize some of the subsequent commercial roles played by individuals active in the open source movement.

4.4 Leadership, organization and governance.

A successful open source project also requires a credible leader or leadership, and an organization consistent with the nature of the process.

Although the leader is often at the origin a user who attempts to solve a particular program, the leader over time performs less and less programming. The leader must (a) provide a vision, (b) make sure that the overall project is divided into much smaller and well-defined tasks ("modules") that individuals can tackle independently from other tasks, (c) attract other programmers, and, last but not least, (d) "keep the project together" (prevent it from forking or being abandoned).

The initial leader must assemble a critical mass of code to which the programming community can react. Enough work must be done to show that the project is doable and has merit. At the same time, to attract additional programmers, it may be important that the leader does not perform too much of the job on his own and leaves challenging programming problems to others.[15] Indeed, programmers will initially be reluctant to join a project whose leadership qualities are yet untested unless they identify an exciting challenge. Another reason why programmers are easier to attract at an early stage is that,

[15] E.g., Valloppillil's [1998] discussion of the Mozilla release.

if successful, the project will keep attracting a large number of programmers in the future, making early contributions very visible.

Consistent with this argument, it is interesting to note that each of the three cases described above appeared to pose challenging programming problems. When the initial release of each of these open source programs was made, considerable programming problems were unresolved. The promise that the project was not near a "dead end," but rather would continue to attract ongoing participation from programmers in the years to come, appears to be an important aspect of its appeal. In this respect, Linux is perhaps the quintessential example. The initial Linux operating system was quite minimal, on the order of a few tens of thousands of lines of code. In Torvalds' initial postings in which he sought to generate interest in Linux, he explicitly highlighted the extent to which the version would require creative programming in order to achieve full functionality.

Another important determinant of project success appears to be the nature of its leadership. In some respects, the governance structures of open source projects are quite different. In a number of instances, such as Linux, there is an undisputed leader. While certain aspects are delegated to others, a strong centralization of authority characterizes these projects. In other cases, such as Apache, a committee will resolve the disputes by voting or a consensus process. At the same time, leaders of open source projects share some common features. Most leaders are the programmers who developed the initial code for the project (or made another important contribution early in the project's development). While many no longer make programming contributions, having moved on to broader project management tasks, the individuals that we talked to believed that the initial experience was important in establishing credibility to manage the project. The

splintering of the Berkeley-derived Unix development programs has been attributed in part to the absence of a single credible leader.

But what does the leadership of an open source project do? It might appear at first sight that the unconstrained, quasi-anarchistic nature of the open source process leaves little scope for a leadership. This, however, is incorrect. First, as we have already indicated, the leadership sets a vision. If the leader is credible and/or the vision is compelling, this vision helps coordinate expectations. Second, even though participants are free to take the project where they want as long as they release the modified code, acceptance by the leadership of a modification or addition provides some certification as to the quality of the latter and its integration/compatibility with the overall project.

As discussed by Max Weber [1968], some attributes underlie a successful leadership.[16] First, the programmers must trust the leadership: that is, they must believe that the leader's objectives are sufficiently congruent with theirs and not polluted by ego-driven, commercial, or political biases. For instance, the leadership must be willing to accept improvements on their merits even though they do not fit the leader's original blueprint.

Trust in the leadership is also key to the prevention of forking. While there are natural forces against forking (the loss of economies of scale due to the creation of smaller communities, the hesitations of programmers in complementary segments to port to multiple versions, and the stigma attached to the existence of a conflict), other factors may encourage forking. User-developers may have conflicting interests as to the evolution of the technology. Ego (signaling) concerns may also prevent a faction from admitting that another approach is more promising, or simply from accepting that it may

[16] See also Hermalin [1998].

socially be preferable to have one group join the other's efforts even if no clear winner has emerged. The presence of a charismatic (i.e., trusted) leader is likely to substantially reduce the probability of forking in two ways. First, indecisive programmers are likely to rally behind the leadership's preferred alternative. Second, the dissenting faction may not have an obvious leader of its own.

A good leadership should also clearly communicate its goals and evaluation procedures. Indeed, the open source organizations go to considerable efforts to make the nature of their decision making process transparent: the process by which the operating committee reviews new software proposals is frequently posted and all postings archived. For instance, on the Apache web site, it is explained how proposed changes to the program are reviewed by the program's governing body, whose membership is largely based on contributions to the project. (Any significant change requires at least three "yes" votes—and no vetoes—by these key decision-makers.)

5. Commercial Software Companies' Reactions to the Open Source Movement

This section examines the interface between open and closed source software development. Challenged by the successes of the open source movement, the commercial software corporations may employ one of the following two strategies. The first is to emulate some incentive features of open source processes in a distinctively closed source environment. Another is to try to mix open and closed source processes to get the best of both worlds.

5.1. *Why don't corporations duplicate the open source incentives?*

As we already noted, owners of proprietary code are not able to enjoy the benefits of getting free programmer training in schools and universities (the alumni effect); nor can they easily allow users to modify their code and customize it without jeopardizing intellectual property rights. Similarly, and for the reasons developed in section 4, commercial companies will never be able to duplicate the visibility of performance reached in the open source world.

In contrast, they can to some extent duplicate some of the signaling incentives of the open source world. Indeed, a number of commercial software companies (e.g., video game companies, Qualcomm for the Eudora email program) list people who have developed the software. It is an interesting question why others do not. To be certain, commercial companies do not like their key employees to become highly visible, lest they be hired away by competitors.[17] But, to a large extent, firms also realize that this very visibility enables them to attract talented individuals and provides a powerful incentive to existing employees.

Another area in which software companies might try to emulate open source development is the promotion of widespread code sharing within the company. This may enable them to reduce code duplication and to broaden a programmer's audience. Interestingly, existing organizational forms may preclude the adoption of open source systems within commercial software firms. An internal Microsoft document on open source [Valloppillil, 1998] describes a number of pressures that limit the implementation of features of open source development within Microsoft. Most importantly, each software development group appears to be largely autonomous. Software routines

[17] For instance, concerns about the "poaching" of key employees was one of the reasons cited for Steve Jobs' recent decision to cease giving credit to key programmers in Apple products [Claymon, 1999].

developed by one group are not shared with others. In some instances, the groups seek to prevent being broken up by not documenting a large number of program features. These organizational attributes, the document suggests, lead to very complex and interdependent programs that do not lend themselves to development in a "compartmentalized" manner nor to widespread sharing of source code.[18]

5.2 *The commercial software companies' open source strategies.*

As should be expected, many commercial companies have elaborated strategies to capitalize on the open source movement. In a nutshell, they expect to benefit from their expertise in some segment whose demand is boosted by the success of a complementary open source program. While improvements in the open source software are not appropriable, commercial companies can benefit indirectly in a complementary proprietary segment.[19]

One such strategy is straightforward. It consists of commercially providing complementary services and products that are not supplied efficiently by the open source community. Red Hat and VA Linux for example, exemplify this "reactive" strategy.[20]

A "reactive" commercial company may still want to encourage and subsidize the open source movement, for example by allocating a few programmers to the open source

---

[18]Cusamano and Selby [1995], however, document a number of management institutions at Microsoft that attempt to limit these pressures.

[19] Another motivation for commercial companies to interface with the open source world may be public relations. We do not view this as a permanent strategy, however, as programmers and consumers presumably cannot be so easily fooled over an extended period. Similarly, firms may temporarily encourage programmers to participate in open source projects to learn about the strengths and weaknesses of this development approach.

[20] Red Hat provides support for Linux-based products, while VA Linux provides hardware products optimized for the Linux environment. In December 1999, their market capitalizations were $17 and $10 billion respectively.

project.[21] Red Hat will make more money on support if Linux is successful. Similarly, if logic semiconductors and operating systems for personal computers are complements, one can show by a revealed preference argument that Intel's profits will increase if Linux (which unlike Windows is free) takes over the PC operating system market. Similarly, Sun may benefit if Microsoft's position is weakened. Oracle may wish to port its database products to a Linux environment in order to lessen its dependence on Sun's Solaris operating system. Because firms do not capture all the benefits of the investments, the free-rider problem often discussed in the economics of innovation should apply here as well. Subsidies by commercial companies for open source projects should remain limited unless the potential beneficiaries succeed in organizing a consortium (which will limit the free-riding problem).

A second strategy is to take a more proactive role in the development of open source software. Companies can release existing proprietary code and create some governance structure for the resulting open source process. For example, Hewlett-Packard recently released its Spectrum Object Model-Linker open source in order to help the Linux community port Linux to Hewlett Packard's RISC architecture. They can even (though probably less likely) encourage "ex nihilo" development of new pieces of open source software. This is similar to the strategy of giving away the razor (the released code) to sell more razor blades (the related consulting services that HP will provide).

Various efforts by corporations selling proprietary software products to develop additional products through an open source approach have been undertaken. One of the most visible of these efforts was Netscape's 1998 decision to make "Mozilla," a portion

[21] Of course, these programmers also increase the company's "absorptive capacity" and help the company with the development of the proprietary segment.

of its browser source code, freely available. This effort has been so far relatively unsuccessful. Indeed, the Mozilla effort only received approximately two dozen postings by outside developers. Perhaps (and we can only conjecture here), the launching failed because it occurred too late (many exciting tasks had already been completed). It is also likely that Netscape did not adopt the right governance structure. Leadership by a commercial entity may not internalize enough of the objectives of the open source community. In particular, a corporation may not be able to credibly commit to keeping all source code in the public domain and to adequately highlighting important contributions.[22]

In this light, it is tempting to interpret the creation of organizations such as Collab.Net as efforts to certify corporate open source development programs, just as investment banks and venture capitalists play a certification role for new firms. Collab.Net, a new venture funded by the venture capital group Benchmark Partners, will organize open source projects for corporations who wish to develop part of their software in this manner. Collab.net will receive fees for its online marketplace (SourceXchange, through which corporations will contact open source developers), for preparing contracts, for helping select and monitor developers, and for settling disputes. Hewlett Packard

---

[22] For instance, in the Mozilla project, Netscape's unwillingness to make large amounts of browser code public was seen as an indication of its questionable commitment to the open source process. In addition, Netscape's initial insistence on the licensing terms that allowed the corporation to relicense the software developed in the open source project on a proprietary basis was viewed as problematic [Hamerly, Paquin, and Walton, 1999]. The licensing terms however may not have been the hindering factor, since the terms of the final license are even stricter than those of the GPL.

released the core of its E-speak technology (which enable brokering capabilities) to open source[23] and posted six projects related to this technology.

Hewlett Packard's management of the open source process seems consistent with Dessein [1999]. Dessein shows that a principal with formal control rights over an agent's activity in general gains by delegating his control rights to an intermediary with preferences or incentives that are intermediate between his and the agent's. The partial alignment of the intermediary's preferences with the agent's fosters trust and boosts the agent's initiative, ultimately offsetting the partial loss of control for the principal. In the case of Collab.net, the congruence with the open source developers is obtained through the employment of visible open source developers (for example, the president and chief technical officer is Brian Behlendorf, one of the cofounders of the Apache project) and the involvement of O'Reilly, a technical book publisher with strong ties to the open source community.

When can it be advantageous for a commercial company to release proprietary code under an open source license? The first condition is, as we have noted, that the company expects to thereby boost its profit on a complementary segment. A second is that the increase in profit in the proprietary complementary segment offsets any profit that would have been made in the primary segment, had it not been converted to open source. Thus, the temptation to go open source is particularly strong when the company is too small to

[23] Some of the E-speak code remains proprietary to Hewlett Packard; so will some applications and utilities developed in the future. It should also be noted that HP can profit by providing services to E-speak users, which, while not proprietary, should be an arena in which HP has a natural advantage.

compete commercially in the primary segment or when it is lagging behind the leader and about to become extinct in that segment.[24]

5.3 Can commercial activities pollute the open source process?

The flexible open source license allows for the coexistence of open and closed source code. While it represents in our view (and in that of many open source participants) a reasonable compromise, it is not without hazards.

First, the open source project may be "hijacked" by a participant who builds a valuable module and then offers proprietary APIs to which application developers start writing. The innovator has then built a platform that appropriates some of the benefits of the project. To be certain, open source participants might then be outraged, but it is unclear whether this would suffice to prevent the hijacking. The open source community would then be as powerless as the commercial owner of a platform above which a "middleware" producer superimposes a new platform.[25]

Second, the coexistence of commercial activities may alter the programmers' incentives. To understand why it may be useful to make an analogy with academia (despite some differences between the academic research and open source development processes).

To put our reflections in perspective, let us first argue against the view that one should prevent academic research from being polluted by outside activities. We believe that these outside activities—e.g., for an economist, work with firms and financial

[24] See, for example, the discussion of SGI's open source strategy in Taschek [1999].

[25] The increasing number of software patents being granted by the U.S. Patent and Trademark Office provide another avenue through which such a "hijacking" might occur. In a number of cases, industry observers have alleged that patent examiners—not being very familiar with the unpatented "prior art" of earlier software code—have granted unreasonably broad patents, in some cases giving the applicant rights to software that was originally developed through open source processes.

intermediaries and participation in the public policy process; for an engineer, consulting with large firms and part-time work in start-ups—provides a useful two-way transfer of knowledge between practical matters and fundamental research. They also provide academics who desire it with (instantaneous and intertemporal) job diversification and thereby increase the attractiveness of academia. Yet, like many researchers, we are concerned that these outside activities may pollute the research process and make it less attractive. There are several reasons for this.

First, a field may be deprived of some of its best minds if they neglect fundamental research and pursue applied, specific-purpose projects rather than broader, general-purpose innovations. The field may lose some of its allure as an exciting intellectual environment in which new insights accrue at a rapid pace and a large community that avidly reads about the latest developments.

Second, the academic process may lose some of its integrity. The high-powered incentives provided by outside activities may induce researchers to sell off some of the long-term reputational capital in the academic community. This happens for instance when a researcher puts his intellectual weight behind a dubious idea, directs students excessively to his or her start-up or consulting firm, rejects papers submitted to scientific journals simply because they do not mesh with the views espoused in the outside activity, or stops freely exchanging knowledge. To be certain, some of these behaviors are already motivated by purely academic incentives. Our point is simply that these may be exacerbated by the presence of powerful outside incentives, and by the shortening of the relationships that results from the move of academics to other communities.

While it is too early to tell, some of these same issues may appear in the open source world. Programmers working on an open source project may be tempted to stop interacting and contributing freely if they think they have an idea for a module that might yield a huge commercial payoff. Too many programmers may start focusing on the commercial side, making the open source process less exciting.

6. What are the Open Economic Questions about Open Source?

There are many other issues posed by open source development that do not lend themselves as neatly to conventional economic frameworks. This section will highlight a number of these as suggestions for future work.

To what extent will reliance on breaking software projects into distinct modules serve to limit the effectiveness of open source? The success of an open source project is dependent on the ability to break the project into distinct components. Without an ability to parcel out work in different areas to programming teams who need little contact with one another, the effort is likely to be unmanageable. Some observers argue that the underlying Unix architecture lent itself well to the ability to break development tasks into distinct components. It may be that as new open source projects move beyond their Unix origins and encounter new programming challenges, the ability to break projects into distinct units will be less possible. But recent developments in computer science and programming languages (e.g., object-oriented programming) have encouraged further modularization, and may facilitate future open source projects.

Can the management of open source projects accommodate the increasing number of contributors? The frequency and quality of contributions to each of the open source

projects studied appears to be highly skewed, with a few individuals (or at most a few dozen) accounting for a disproportionate amount of the contributions, with most programmers making just one or two submissions. Many contributors to Sendmail, for instance, are students, who use the open source project as part of their software engineering training. These claims are corroborated by the assessment of over 4000 contributions to a Linux development site by Dempsey, *et al.* [1999], who conclude that developers of more than two Linux applications, while accounting for only 9% of the overall sample, account for over 38% of the total contributions. If large numbers of low-quality contributions are becoming increasingly common, there may be substantial management challenges in the future.[26] In this setting, sorting though the large number of software submissions of varying quality is likely to be an increasingly onerous task, one that will swamp the efforts of a volunteer staff. One possibility is the model being adapted by Sendmail, where professional employees of the for-profit firm manage the open source submissions (as well as making their own contribution). But in general, the extent to which individual open source projects can accommodate substantial growth in contributors remains an open question. (At the same time, it must be acknowledged that the same skewness of output is also observed among programmers employed in commercial software development facilities [e.g., see Brooks, 1975].)

To what extent do open source projects have a greater or shorter effective life span than traditional projects? One of the arguments offered by open source advocates is that because their source code is publicly available, and at least some contributions will continue to be made, its software will have a longer duration. (Many software products

[26] Linus Torvalds has succeeded in overseeing the Linux process by delegating to trusted lieutenants who themselves have sub-delegated.

by commercial vendors are abandoned or no longer upgraded after the developer is acquired or liquidated, or even when the company develops a new product to replace the old program.) But another argument is that the nature of incentives being offered open source developers—which as discussed above, lead them to work on highly visible projects—might lead to a "too early" abandonment of projects that experience a relative loss in popularity. An example is the XEmacs project, an open source project to create a graphical environment with multiple "windows" that originated at Stanford. Once this development effort encountered an initial decline in popularity, many of the open source developers appeared to move onto alternative projects.

Our ability to answer confidently these and related questions is likely to increase as the open source movement itself grows and evolves. At the same time, it is heartening to us how much of open source activities can be understood within existing economic frameworks, despite the presence of claims to the contrary. The literature on "career concerns" provides a lens through which the structure of open source projects, the role of contributors, and the movement's ongoing evolution can be viewed.

References

Brooks, Fredrick, 1975, *The Mythical Man Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley.

Browne, Christopher B., 1999, "Linux and Decentralized Development," http://www.firstmonday.dk/issues/issue3_3/browne/index.html (accessed December 17, 1999).

Claymon, Deborah, 1999, "Apple in Tiff with Programmers over Signature Work," *San Jose Mercury News.* December 2, 1999.

Cusumano, Michael A., and Richard W. Selby, 1995, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York: Free Press.

Darwall, Christina, and Josh Lerner, 2000, "A Note on Open Source Software Development," Harvard Business School Case No. 200-___.

Dempsey, Bert J., Debra Weiss, Paul Jones, and Jane Grenberg, 1999, "A Quantitative Profile of a Community of Open Source Linux Developers," Unpublished working paper, School of Information and Library Science, University of North Carolina at Chapel Hill, http://metalab.unc.edu/osrt/develpro.html (accessed November 1, 1999).

Dessein, Wouter, 1999, "Authority and Communication in Organizations," Unpublished working paper, Université Libre de Bruxelles.

DiBona, Chris, Sam Ockman, and Mark Stone, editors, 1999, *Open Sources: Voices from the Open Source Revolution.* Sebastopol, California: O'Reilly.

Ghosh, Rishib A., 1999, "Cooking Pot Markets: An Economic Model for the Trade in Free Goods and Services on the Internet," http://www.firstmonday.dk/issues/issue3_3/ghosh/index.html (accessed December 17, 1999).

Gomulkiewicz, Robert W., 1999, "How Copyleft Uses License Rights to Succeed in the Open Source Software Revolution and the Implications for Article 2B," *Houston Law Review*. 36, 179-194.

Hammerly, Jim, Tom Paquin, and Susan Walton, 1999, "Freeing the Source: The Story of Mozilla," in Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. Sebastopol, California: O'Reilly, pp. 197-206.

Hermalin, Benjamin E., 1998, "Toward an Economic Theory of Leadership: Leading by Example," *American Economic Review*. 88, 1188-1206.

Holmström, Bengt, 1999, ``Managerial Incentive Problems: A Dynamic Perspective," *Review of Economic Studies*. 66, 169-182.

Johnson, Justin P., 1999, "Economics of Open-Source Software,'' Unpublished working paper, Massachusetts Institute of Technology.

Levy, Steven, 1984, *Hackers: Heroes of the Computer Revolution.* Garden City, NY: Anchor Press/Doubleday, 1984.

Nadeau, Tom, 1999, "Learning from Linux," http://www.os2hq.com/archives/linmemo1.htm (accessed November 12, 1999).

Open Source Initiative, 1999, "Open Source Definition," http://www.opensource.org/osd.html (accessed November 14, 1999).

Raymond, Eric S., 1999a, "The Cathedral and the Bazaar," http://www.tuxedo.org/-esr/writings/catherdral-bazaar (accessed November 9, 1999).

Raymond, Eric S., 1999b, "Homesteading the Noosphere: An Introductory Contradiction," http://www.tuxedo.org/-esr/writings/homesteading (accessed November 11, 1999).

Rosenberg, Nathan, 1976, *Perspectives on Technology*. Cambridge: Cambridge University Press.

Stallman, Richard, 1999, "The GNU Operating System and the Free Software Movement," in Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. Sebastopol, California: O'Reilly, pp. 53-70.

Taschek, John, 1999, "Vendor Seeks Salvation By Giving Away Technology," http://www.zdnet.com/pcweek/stories/news/0,4153,404867,00.html (accessed December 17, 1999).

Torvalds, Linus, 1999, "Interview with Linus Torvalds: What Motivates Free Software Developers?," http://www.firstmonday.dk/issues/issue3_3/torvalds/index.html (accessed December 17, 1999).

Valloppillil, Vinod, 1998, "Open Source Software: A (New?) Development Methodology," [also referred to as The Halloween Document], Unpublished working paper, Microsoft Corporation, http://www.opensource.org/hallowen/halloween1.html (accessed November 9, 1999).

von Hippel, Eric, 1988, *The Sources of Invention.* New York: Oxford University Press.

Weber, Max, 1968, *Economy and Society: An Outline of Interpretative Sociology.* (Guenther Roth and Claus Wittich, editors). New York: Bedminster Press.

Table 1: The three open source programs studied

Program	*Apache*	*Perl*	*Sendmail*
Nature of program:	World wide web (HTTP) server	System administration and programming language	Internet mail transfer agent
Year of introduction:	1994	1987	1979 (predecessor program)
Governing body:	Apache Software Foundation	Selected programmers (among the "perl-5-porters") (formerly, The Perl Institute)	Sendmail Consortium
Competitors:	Internet Information Server (Microsoft)	Java (Sun)	Exchange (Microsoft)
	Various servers (Netscape)	Python (open source program)	IMail (Ipswitch)
		Visual Basic, ActiveX (Microsoft)	Post.Office (Software.com)
Market penetration:	55% (September 1999) (of publicly observable sites only)	Estimated to have 1 million users	Handles ~80% of Internet e-mail traffic
Web site:	www.apache.org	www.perl.org	www.sendmail.com

Table 2: Commercial roles played by selected individuals active in open source movement

Individual	*Role and Company*
Eric Allman	Chief Technical Officer, Sendmail, Inc. (support for open source software product)
Brian Behlendorf	er, President, and Chief Technical Officer, Collab.Net (management of open source projects)
Keith Bostic	Founder and President, Sleepycat Software
L. Peter Deutsch	Founder, Aladdin Enterprises (support for open source software product)
William Joy	Founder and Chief Scientist, Sun Microsystems (workstation and software manufacturer)
Michael Tiemann	Founder, Cygnus Solutions (open source support)
Linus Torvalds	Employee, Transmeta Corporation (chip design company)
Paul Vixie	President, Vixie Enterprises (engineering and consulting services)
Larry Wall	Employee, O'Reilly & Associates (software documentation publisher)